Approximate Volume Rendering for Curvilinear and Unstructured Grids by Hardware-assisted Polyhedron Projection

Nelson Max[1], Peter Williams[1], and Claudio Silva[2]

[1] Lawrence Livermore National Laboratory

[2] AT&T Laboratory-Research

Abstract

A hardware polygon rendering pipeline can be used together with hardware compositing to volume-render arbitrary unstructured grids composed of convex polyhedral cells. This technique is described, together with the global sorting necessary for back-to-front compositing, and the modifications that must be made to approximate curvilinear grids, whose faces may not be planar.

1. Introduction

There are many situations where it is useful to visualize, in a 2D image, a quantity like density or pressure that varies within a 3D volume. Applications include medical data from ultrasound, MRI or X-ray tomography, seismic data, or data computed in numerical simulations in mechanics, hydrodynamics, or aerodynamics. For volume rendering, the quantity to be visualized is transformed into variations in the color and opacity of a semitransparent glowing substance, and the 2D image is a view of this glowing volume. For the mathematics behind this optical model, and other more sophisticated models, see Max[1].

The data from medical images or finite difference simulations is defined at the vertices of a regular cubic or rectangular grid, which makes volume rendering easier. However, measurements of real world data may only be available at irregularly scattered locations. Irregular finite elements may be needed to fit the shape of complex objects, or to concentrate small elements near a shock where pressure varies rapidly. So it is useful to have volume rendering algorithms that can accept arbitrary "unstructured" grids of convex polyhedral elements. Here we concentrate on the "zoo elements" commonly used in finite element grids, which have the topology (face-edge-vertex connectivity), but not necessarily face shapes of, a cube (hexahedron), tetrahedron, triangular prism, or square pyramid. All faces of such zoo elements are either triangles or quadrilaterals.

One method for volume rendering is to slice the volume by equally spaced planes perpendicular to the viewing direction. The color at each point (or *pixel*) in the 2D raster image is deter-

mined by the colors and opacities in the slice planes, at points along a viewing ray through the pixel. One can compute the final image by *compositing* the slice planes in back to front order. To include the effect of a new slice plane, multiply each old pixel color by the transparency (one minus the opacity α) of the corresponding pixel in the new slice plane, and then add the color contributed by the new slice pixel's glow. (See equation (1) of section 2 below.) In this way, the colors from the more distant parts of the volume are partially hidden by the opacity in front.

In a cubic grid volume, the slice planes will be at a slant for most viewing directions, so the color and opacity at a slice plane pixel must be interpolated from the eight nearest grid points. Hardware to do this interpolation and compositing arithmetic is available on current visualization workstations with 3D texture mapping [2]. The flow of data is more coherent if the slices are taken along the data planes, which are not usually perpendicular to the view direction, and hardware to do this variant is available on PC cards [3].

One must resample an irregular grid into a cubic grid to use these hardware techniques, and this can greatly expand the amount of data. An adaptive computational grid will have small cells in regions where the scalar variable changes rapidly, for example, near an airplane wing or a shock, and much larger cells where the variable changes slowly and/or smoothly. A cubic grid fine enough to capture the detail from the small cells will waste many cubes inside the large cells. In addition, the resampling takes time, and may blur extreme values and rapid variation. Below, we concentrate on methods which render the irregular grid directly, without resampling.

Slicing an unstructured grid of polyhedral cells is possible using software together with hardware. A slice plane intersects a cell in a polygon, whose shape can be computed incrementally from the corresponding polygon in the previous slice [4]. The color and opacity can be interpolated in software from the cell vertices to the slice polygon vertices. They can them be interpolated across the slice polygon by standard polygon rendering and compositing hardware.

An alternative for software alone is to do the computation per pixel, rather than per slice. The sample points in each slice plane which affect an image pixel lie along a straight line, the "viewing ray", so this method is often called "ray tracing". The color and opacity are interpolated at the sample points on the ray, and the compositing is performed for the single pixel. Ray tracing is easy for a rectangular grid, because the indices of the cell containing each sample on the viewing ray can be directly computed from the geometry of the ray and the data volume. For irregular grids, one must compute the sequence of segments in which the ray intersects successive grid cells [5]. Given the face through which a ray leaves one cell, one can find the next cell it enters using data on the topological connectivity of the grid. Then one determines the face through which it exits that cell, and so forth. It is usually possible to compute the color and transparency of a ray segment in a cell by analytic integration, rather than by taking sample points [6, 7]. So the color of the image pixel for the ray can be found by compositing the segments, rather than the slice samples, saving computation if the cells are large.

The *polyhedron projection* algorithm can speed up this process by compositing a whole cell at once, taking advantage of area coherence within the cell faces. The projections of all the edges of the cell divide the image plane into a number of polygonal regions, as in figure 1. In each such region, all the viewing rays through image pixels intersect the same two front and back planar faces, so the thickness varies linearly. Therefore, if the colors and opacities for the viewing rays through the region's vertices are computed by analytic integration, they can be interpolated and composited by polygon rendering hardware. Details of this technique are given in section 2 below.

The cells must be sorted in back to front order for this compositing to be correct. Imagine a pile of cells on a table, being viewed from far beneath the table. In this case, a back to front sort is top to bottom. Cells are removed one by one from the top of the pile, and composited into the image. A cell can be safely removed only if can be lifted vertically without disturbing any other cells which lie on top of it. This is because any cell on top should have been already removed and composited, in order to be correctly affected by the opacity of the current cell.

This is like the game of "pick up sticks". The sticks are the cells, and they may fall such that each stick is beneath another stick, and no stick can be removed first. We call such a situation a *visibility cycle*, because it must contain a periodic cycle of sticks, each beneath the next one in the cycle. In such a case, a back to front sort is only possible if one or more of the cells is cut in two. Such problems rarely occur in practical grids. But it is still a challenging problem to compute a good sorting order efficiently. Before removing a cell, an unsophisticated algorithm would test for overlap with every other remaining cell, so the sorting time would grow as the square of the number of cells, and be very slow for large grids. In section 3 below, we discuss more efficient ways of sorting the cells.

In the above discussion of unstructured grids, we have assumed the cells are convex polyhedra, with planar faces. But this excludes simulation data from curvilinear grids, which have rectangular grid topology, but are deformed to match curved geometry, such as the wing of an airplane. Also, initially planar quadrilateral faces on a finite element grid may become non-planar as the grid deforms during a simulation.

Such non-planar quadrilateral faces cause problems with both the cell projection and cell sorting algorithms. Figure 2 shows a view of two hexahedra, in which a viewing ray enters cell *A*, exits through a curved face into cell *B*, and then renters cell *A* again through the same face. Here A and B form a two-element visibility cycle, and no back to front sort is possible. The solution is to cut cell *A* into two or more pieces, and the simplest way of doing this is to divide it into tetrahedra. Each quadrilateral face of cell *A* will be divided by a diagonal into two triangles. If cell *B* is also divided into tetrahedra, it is important to choose a consistent diagonal for their common face, so that there is no gap or overlap between the two approximated cells. Section 4 below gives a way of doing this. We only subdivides cell intersected twice by a viewing ray, and project other cells as a whole, even though they may have non-planar faces. Cells containing rendered contour surfaces,

or discontinuities in the color or opacity, may also be subdivided. Section 5 describes how we do this view-dependent subdivision into tetrahedra. It gives only an approximation to the correct volume rendering for curvilinear finite elements, but can be achieved effeciently using current hardware.

2. Polyhedron Projection

To project a polyhedron is to compute its color and opacity contributions to all viewing rays that intersect it, and composite them onto the accumulating image. Consider the convex polyhedron *P = ABCDEF* whose projection on the image plane is shown in figure 1. The projected edges divide the image plane into five polygonal regions for hardware rendering and compositing: triangles A*BG*, *BCG*, and *DEF*, and quadrilaterals *AGED* and *GCFE*.

Several authors have considered such subdivisions for hardware rendering. Shirley and Tuchman [8] constructed the four planar subdivision topologies that can arise from projecting a tetrahedron, and Wilhelms and Van Gelder [9] used a plane sweep algorithm to construct the subdivision for any projection of a cube. In our system, we construct a winged-edge [10, 11] subdivision structure for the projection of an arbitrary polyhedron *P*. We add the projected edges of *P* one by one to this structure, by moving along a new edge and finding its intersections with existing edges. A new vertex is created at each intersection (unless it coincides with an existing vertex) and the winged-edge structure is modified, perhaps further dividing one of the existing polygons in two.

Each polygon *Q* in the final subdivision of the image plane lies in the projection of one front face and one back face of *P*. If the viewing rays are parallel, for a viewpoint "at infinity", the thickness of *P*, that is, the length of the viewing ray segment in *P*, varies linearly over *Q*. (For perspective, we also assume it varies linearly over *Q*, which is a good approximation for small or distant cells.) The thickness is zero at profile vertices like vertex *A* in figure 1, and can be computed from the geometry of *P* and of the viewing ray at non-profile vertices like *E*, and projected edge intersections like *G*.

In [8] and [9], the opacities along the rays through the vertices of a screen polygon *Q* are interpolated linearly (or bilinearly) by the graphics hardware. However, we will analyze the opacity effects below, and show that the opacity is a non-linear function of the thickness.

The volume opacity in volume rendering actually represents the *extinction coefficient* τ, which expresses the infinitesimal fractional decrease in radiance (or light intensity) *I*, per unit infinitesimal length *ds* along a viewing ray. (See [1].) Thus the intensity satisfies the differential equation

$$\frac{dI(s)}{ds} = -\tau(s)I(s) \quad ,$$

or

$$\frac{dI(s)}{I(s)} = -\tau(s)ds.$$

Integrating along a ray segment of length l,

$$\ln I(l) - \ln I(0) = -\int_0^l \tau(s)ds,$$

or, taking exponentials,

$$I(l) = I(0)\exp\left(-\int_0^l \tau(s)ds\right).$$

The quantity $\exp\left(-\int_0^l \tau(s)ds\right)$ represents the transparency of the volume along the ray segment. If τ is constant inside the cell volume, the transparency reduces to $\exp(-l\tau)$, which clearly varies non-linearly with l. Thus correct volume rendering requires an exponential for each pixel. If linear or bilinear interpolation of vertex transparency is used instead, distracting Mach bands can result, as shown in Stein *et al*. [12].

Correct volume rendering is still possible in current hardware if a texture map is available to get the exponential per pixel with a table look up. For constant τ, the product $l\tau$ is used as a 1D texture coordinate, and for varying τ, l and τ are used as a 2D texture coordinate pair.

Texture mapping was originally developed to enhance the apparent detail of a polygonal model, without increasing the polygon count. The additional detail is stored in a rectangular raster image, which is addressed by two texture coordinates (u, v). These are specified at the vertices of a polygon, and interpolated in hardware across the polygon in the same way as a color or depth value would be. When the polygon is rendered, the texture at the interpolated address is accessed and combined with the interpolated color and opacity to determine the final effect on the image. Although the texture map is rectangular, the textured polygon need not be. Its image in the texture map is determined by the texture coordinates specified at the polygon vertices, and in general will be a subset of the texture map rectangle.

The OpenGL specifications [13] are non-committal on how vertex values should be interpolated across polygons, but mention two possibilities: (*a*) divide the polygon up into triangles, and interpolate linearly on each triangle (piecewise linear interpolation), or (*b*) interpolate linearly along the polygon edges, and then linearly across horizontal scan line segments between pairs of edges (piecewise bilinear interpolation). Both of these methods reduce to linear interpolation when the values specified at the vertices are consistent with a single linear function. This is the case for the thickness l across an image plane subdivision polygon Q, but not for the average glow color, to be discussed later.

For constant τ, the vertex values of $l\tau$ are scaled into the range appropriate for an address u into the 1D texture table. The texture table must be preloaded at address u with the quantity 1. - exp(-u), because the OpenGL compositing equation (1) below uses α = 1. - transparency. Care must be taken in scaling the address and table size to get a substantial range of transparencies, and to clamp the address at the ends of its range, instead of letting it "wrap around". The output α of the texture table is used in the compositing equation

$$new\_pixel\_color = \alpha * polygon\_color + (1. - \alpha) * old\_pixel\_color \qquad (1)$$

which multiplies the intensity *old_pixel_color* of the cells composited so far by the appropriate per-pixel transparency 1. - α = exp(- $l\tau$).

Think of the cell volume as filled with small opaque particles glowing with color intensity *polygon_color.* (See [1].) Then the transparency 1. - α means that a viewing ray gets through with probability 1. - α without hitting a particle, and therefore hits a particle with probability α, in which case it sees color *polygon_color.* Thus the above compositing formula includes the glow color with the correct weight.

The glow color and extinction coefficient (often called "opacity") are usually specified by *transfer functions* of the scalar variable being visualized. For now, assume that these transfer functions are linear. If the scalar variable is constant within each cell, it is appropriate to assume, as above, that the color and extinction coefficient are constant in the cell. However for "linear" elements, the variable is specified at the element vertices, and interpolated across the element, so the color and extinction coefficient must also be interpolated. For tetrahedra, linear interpolation suffices, but for other element shapes, more complicated interpolation is required, using the basis functions specified as part of the finite element model. For example, trilinear interpolation would be used on cube-like elements. For the purposes of hardware volume rendering, it is necessary to accept the interpolation scheme implemented in the graphics hardware, even if it does not agree with that used in the finite element analysis.

Assume that the extinction coefficient τ has been specified at the vertices of the polyhedron *ABCDEF* in figure 1, and is interpolated across the faces by the polygon scan-conversion hardware. This will give, for each viewing ray segment in *P*, a front face value τ_f and a back face value τ_b. Assume further that τ is linearly interpolated between τ_f and τ_b along each ray segment. (For a tetrahedron, this scheme will agree with linear interpolation, but for a general cell, it will not usually agree with the interpolation used in the finite element analysis. For example, if the hardware uses the interpolation method (*b*) above, it will correspond to a particular view-dependent piecewise trilinear interpolation.) Then

$$\int_0^l \tau(s)\,ds = \int_0^l \left(\frac{s}{l}\tau_f + \left(1. - \frac{s}{l}\right)\tau_b \right) ds$$

$$= \frac{l^2}{2l}\tau_f + \left(l - \frac{l^2}{2l}\right)\tau_b = l(\tau_f + \tau_b)/2 \ .$$

We can compute the average $\tau_a = (\tau_f + \tau_b)/2$ at each vertex of Q, scale it appropriately, and use it as a second texture coordinate v, which the hardware will interpolate across the polygon Q. The first texture coordinate u is a scaled version of the thickness l. The 2D texture map is loaded with 1. -exp(-uv). Again, care must be taken in the scaling, but since only the product $l\tau_a$ is important, a joint decision on scaling l and τ_a can be taken for each polygon Q, to make best use of the range of exponentials in the table.

The computation of τ_f and τ_b for a vertex of Q depends on the type of the vertex. At profile vertices like A in figure 1, which belongs to both a front facing and a back facing polygon, $\tau_f = \tau_b = \tau_A$, the value specified at A. For intersection vertices like G, τ_f is linearly interpolated in software along edge EB, and τ_b is linearly interpolated along edge AC. For a non-profile vertex like E on the front of P, $\tau_f = \tau_E$, and τ_b is (piecewise) linearly or bilinearly interpolated in software from the values at vertices A, C, F, and D. Non-profile vertices on the back of P are treated similarly.

The glow colors can be interpolated similarly, using software to compute *polygon_color* as the average of the front and back glow colors at the vertices of Q, and then shading hardware to interpolate it across the polygon Q. However, if both the color and extinction coefficient are interpolated linearly across a ray segment in P, the term α**polygon_color* in equation (1) will not give the correct color for the glowing particle optical model. To see this, consider the case that the glow color is red at the front of the ray segment, and green at the back, and the extinction coefficient is a substantially large constant, so that the viewing ray usually hits a particle before travelling very far into the volume. Then the average color will be yellow, but the correct color contributed by the ray segment will be closer to red, because most viewing rays will not get past the red particles. This is equivalent to saying that the opacity of the red region mostly obscures the green region.

In [6] and [7] we show that the correct color can be found by an analytic integral, but its calculation is beyond the per-pixel capabilities of current hardware rendering pipelines. For hardware rendering, we approximate the correct color at every pixel by computing it in analytically software at the non-profile vertices like E and G in figure 1, and dividing it by α to get the *polygon_color* appropriate for use in the compositing equation (1). At the profile vertices, we just set *polygon_color* to the glow color specified at that vertex. Then we use the hardware procedures described above. This at least partially accounts for the effects of extincttion on the color, and does produce the correct α.

3. Sorting

The polyhedron projection method for volume rendering requires a back to front visibility

sort of the cells, such that if cell *A* partially obscures cell *B* from the viewpoint *V*, then *B* comes before *A* in the sorted list, so that the compositing formula (1) correctly accounts for the effect of cell *A*'s opacity on the visibility of cell *B*. In this section, we discuss various algorithms to produce such a visibility sort.

The simplest case is when the grid covers a convex volume with convex polyhedral cells, and comes equipped with an adjacency structure telling, for each face of a cell, which cell, if any, is on the other side of that face. This adjacency structure can be interpreted as a graph, with a node for every cell, and an edge for every pair of cells *A* and *B* sharing a common face *F*. Given a viewpoint *V* for the current view, this graph is turned into a directed graph, whose edge directions depend on the position of the viewpoint *V*. The edge between cells *A* and *B* is directed from *A* to *B* if *V* is on *B*'s side of the plane of their common face *F*, so that *B* obscures *A*, and from *B* to *A* otherwise. Thus in the example of the sticks on the table, the directed edges point down, towards the viewpoint below the table.

We basically do a topological sort of this directed graph, taking time O(*n*), where *n* is the number of cells. (Actually, the sorting time also depends on the number of edges in the directed graph, which is the number of faces in the grid, but for the zoo elements we are considering, each cell has at most 6 faces, so their total number is also O(*n*).)

For the topological sort, we first do a pass through all the cells, counting the number of incoming directed edges. Cells with count zero do not obscure any other cells, and are placed on a queue to be projected. We then remove cells one by one from this queue, and (*a*) for each outgoing directed edge, decrement the count for the cell at the other end of the edge, and if its count becomes zero, put it on the queue, and (*b*) project the cell just removed (or put it next on the sorted list for later projection). This algorithm terminates when the queue becomes empty. At this point, if any cells remain unprojected, they must be involved in a visibility cycle, and some must be subdivided.

One nice feature of this sorting method is that if the geometry does not change between views, the adjacency graph and the plane equations for the faces can be reused. Only the directions of the edges in the adjacency graph need to be revised for a new viewpoint *V*, by evaluating the plane equation of each face *F* at *V*, and using the sign of the result to set the edge direction in the graph.

Edelsbrunner [14] has proved that for a Delaunay tetrahedralization of a set of scattered data points, no visibility cycles are possible. A tetrahedron *A* connecting four of the data points belongs to the Delaunay tetrahedralization only if its circumscribing sphere contains no other data points. Delaunay tetrahedralizations are favored for finite element simulations, because they avoid long thin tetrahedra, which may cause numerical problems. They always define a convex grid of convex cells, filling in the convex hull of the data points. Define the power distance of the view-

point V to the tetrahedron A to be $d^2 - r^2$, where d is distance from V to the center C of A, and r is the radius of its circumscribing sphere. Edelsbrunner's proof is based on the fact that a decreasing sort on this power distance is a visibility sort for a Delaunay tetrahedralization, as can be shown using simple analytic geometry and the condition above on the circumscribing spheres. This power sort is believed to be a good approximate sort for other tetrahedral data sets. Although the time for sorting based on the power distance key is theoretically $O(n \log n)$, it is in practice competitive with the topological sort, because it does not need to store and access the adjacency structure.

The topological sort works for a convex grid because any time a cell A obscure part of cell B from viewpoint V, there is a viewing ray leaving cell A and reaching cell B after crossing a sequence of interior faces of the grid, all represented by edges in the graph. So the directed graph forces B to be projected before A. This is not true if the grid contains cavities or concavities. Then a viewing ray could exit across an exterior face with no directed edge, and then enter the grid again across another exterior face, so that the sequence of directed edges from A to B is broken.

The topological sort can be extended to cover these more general cases by adding directed edges between two boundary cells (those with exterior faces) whenever there is a viewing ray crossing them both. If there are b boundary cells, there can be up to $O(b^2)$ such extra directed edges added to the graph. Using this fact, it is possible to implement a topological sorting technique that takes $O(n + b^2)$ time. Silva *et al*. [15], show how to improve the running time to an output-sensitive bound of $O((b+I) \log^2 b + n)$, where I is the number of crossings among edges of the exterior faces.

In a more recent paper along these lines, Comba *et al*. [16], propose the faster BSP-XMPVO technique based on constructing a binary space partition (BSP) tree on the set of exterior faces of the mesh. Each node of the BSP tree corresponds implicitely to a convex region of space, bounded by planes. Each internal node of the BSP tree stores the plane of one of the exterior faces, which divides the node's region, and the set of exterior faces, into subsets for its two child nodes. The tree is constructed recursively, dividing by these planes until the interior of each leaf node's region contains no exterior faces. Then, given a new viewpoint, the tree can be used to find a visibility sort of the exterior faces, by evaluating the plane equation for the current internal node at the viewpoint V, and recursively proceeding first to the child whose region does not contain the viewpoint. This sort of the exterior faces is combined with the topological sort of the cells to give a correct global sort. As with the topological sort, the BSP tree is unchanged if only the viewpoint moves.

Unfortunately, there may be some *partially projected* cells, which arise as a complication in the method of merging the BSP tree sort with the topological sort. The number p of these is usually very small, in the range of 0.1% to 0.3% of the total number of cells. Each of these p cells

must be tested for possible occlusion with each of the b boundary cells. Thus, once the BSP tree has been constructed, the sorting time for a new viewpoint is $O(n + b\ p)$. We used this BSP-XMPVO sort in the work described in section 5.

A related approach is to separately sort the boundary cells. Williams [17] has shown that an approximate $O(f \log f)$ sort of the f cells with front-facing exterior faces can be combined with the traversal of the directed graph to give an approximate sort for non-convex grids.

A final approach is to fill the space between a grid and its convex hull with e extra convex cells, which are used only for sorting, and are not rendered. Then the topological sort applies, and takes time $O(n + e)$. We are currently exploring this approach. Our goal is to keep the number e of extra cells small, and to generate them efficiently. We also hope not to introduce extra visibility cycles, although our current BSP-based scheme may do so.

Even the brute force $O(n^2)$ visibility sort of n cells can be made more efficient by reducing the constant in front of n^2. The step that is repeated $O(n^2)$ times is the test to see whether cell A can obscure part of cell B from a viewpoint V. This same test is used to add the extra directed edges discussed above, between exterior cells. It is also used in the BSP-XMPVO algorithm of [16], to test each of the p partially projected cells to see whether is obscured by any of the b boundary cells.

To make this test more efficient, Newell *et al*. [18] divide it up into a sequence of simpler tests, any one of which may determine that A does not obscure B. They are ordered in increasing difficulty, so that the easy tests may answer the question before the more difficult ones need be applied. One such scheme is to precompute a bounding box for each cell, that is, to determine its extent along the x, y, and z directions, assuming the viewing is direction along the z axis. Then the tests in order are:

1) The z extent of B is in front of the z extent of A.

2) The x extents of A and B do not overlap on the image plane.

3) The y extents of A and B do not overlap on the image plane.

4) All vertices of B lie in front of the plane of one of the front facing faces of A.

5) All vertices of A lie in back of the plane of one of the back facing faces of B.

If bounding spheres are prepared for each cell, tests 2) and 3) can be replaced or supplemented by

2') The cones from the viewpoint to the bounding spheres of cells A and B are disjoint.

If none of these tests eliminate the possibility of A obscuring B, Newell [19] proposed a definitive search for a separating plane between the cells, using linear programming. Stein *et al*.

[12] (see [7] for a correction) instead searched for an intersection between the projections of an edge of *A* and edge of *B*. The ray through such an intersection will pass through both cells, and can thus be used to determine which cell is in front.

The overlaps discovered in the above tests can be organized into a directed graph, and used in the topological sort. Alternatively, they can be organized into the sorting algorithm of Newell *et al*. [18, 19] described below.

A straightforward use of the above tests would apply at least the first test for every ordered pair of cells. However, there are ways to eliminate large collections of pairs from consideration. Newell's algorithm starts with an initial O(n log n) back to front sort of the cells, using the z of their farthest vertex. The farthest cell *A* is taken from the list and tested with the other cells in order, using the sequence of tests above. As soon as a cell *B* is found that satisfies condition 1), we know that the search for obscured cells can be terminated early. If no obscured cell is discovered, *A* is removed form the list, and projected. If an obscured cell *B* is discovered, it is moved to the head of the list, and tests begin with it instead. This can destroy the z sort needed for the early termination of the testing, so we must keep track of the position in the list after which this order is still undisturbed and early termination remains valid.

If there is a visibility cycle, the above process could lead to an infinite loop, in which the same cells are moved again and again to the head of the list. Therefore, the first time a cell is moved, it is marked as moved. If it is about to be moved a second time, we know it is involved in a cycle, and must be subdivided.

Another way of eliminating pairs of cells from occlusion testing, which is compatible with the early termination method above, is given in Williams *et al*. [7]. Divide the image plane up into windows, and prepare a farthest-z-sorted list of the cells overlapping each window. Then for a candidate cell *A*, we need only check the lists for those windows that cell *A* overlaps.

4. Subdividing Zoo Elements into Consistent Tetrahedra

The above sorting and projection algorithms both assume the cells are convex polyhedra, but as mentioned at the end of the introduction, deformed grids often have non-planar quadrilateral faces. As shown in figure 2, a *problem face* is a quadrilateral which, when approximated by subdivision into two triangles, does not project in a one-to-one way. This causes a visibility cycle due to the non-convex cell *A*. Our solution is to subdivide non-convex *problem cells* like *A* into tetrahedra, which have only planar triangular faces. When doing this subdivision, it is important to choose a consistent diagonals on any shared quadrilateral face, in order to avoid a gap or overlap between the cells on either side of the face. A method for such subdivision was developed for a more specific purpose by Nielson and Sung [20]. Here is a simplification of their method, which we discovered independently, for the problem of subdividing some or all of the zoo elements in a

grid into tetrahedra using consistent diagonals on the quadrilateral faces.

We assume a linear order on the set of vertices, which would arise from the assignment of a different integer index to each vertex. In practice vertices are referenced by such indices in order to avoid repeating coordinates and data for vertices which are shared by several cells. Using this linear order, our decision rule chooses the diagonal for each quadrilateral that starts from its smallest vertex. Since this decision involves only the four vertices of the quadrilateral, it is taken the same way for two adjacent cells that share a quadrilateral face, thus assuring consistency without the use of any mesh connectivity pointers. Below, we show that this choice of diagonals is consistent with a tetrahedral subdivision of each pyramid-, prism-, or cube-like cell.

A pyramid can be divided into two tetrahedra by the plane through either diagonal of the quadrilateral base and the vertex opposite the base, so the diagonal from the smallest base vertex will work.

A triangular prism will have a unique smallest vertex L in the linear order, and the two quadrilateral faces that share L will have diagonals starting at L. The plane through these two diagonals divides the prism into a tetrahedron and a pyramid. As above, the pyramid can be divided using a diagonal of its quadrilateral base, so we get three consistent tetrahedra in all.

For the cube shown in figure 3, assume that vertex A is the smallest in the linear order. Then the three quadrilaterals meeting at this vertex will have diagonals starting there, as shown in the figure. We are not assuming anything about the order of the other seven vertices, so we must show that each of the eight possible sets of choices for the diagonals of the remaining three faces produces a configuration that can be divided into consistent tetrahedra. For each such set of diagonals, count the number that have the vertex G farthest from A as one of their endpoints. If the number is zero, then the three diagonals in this set, plus the three diagonals already shown in figure 2, are the six edges of a central tetrahedron. When this tetrahedron is removed, four other tetrahedra remain, making a total of five.

If there is at least one diagonal with vertex G as an endpoint, then this diagonal, together with the parallel face diagonal starting at vertex A, defines a quadrilateral slice, dividing the hexahedron into two triangular prisms. (For example, if the diagonal DG is present, the slice is quadrilateral $ADGF$.) Since vertex A is the smallest vertex of the hexahedron, apply the decision rule to choose diagonal AG for this slice quadrilateral. Then, as shown above, each prism can be divided up into three consistent tetrahedra, making six tetrahedra in all.

The above proof contradicts part of [21]. The cube in figure 3 is oriented the same as that in figure 3 of [20]. The three diagonals shown in figure 3 correspond to a zero bit for the first, third, and fifth rows of table 1 in [21]. Therefore the eight sets of choices discussed above correspond to six bit codes 0x0y0z, where x, y, and z can be either 0 or 1. In table 2 of [21], cases 010001 and

010100 are listed as subdividable into two prisms, but it is claimed that the two prisms are forced to have inconsistent diagonals across their common quadrilateral face. However, the above proof shows that diagonal *AG* (corresponding to 1-to-7) is a consistent diagonal for this slicing quadrilateral.

5. View-Dependent Subdivision

Subdividing all hexahedral elements in a grid completely into tetrahedra could multiply the number of cells by as much as six, and make sorting and projection take longer. For example, the hexahedron shown in figure 4*a* divides the image plane into 5 quadrilaterals and 2 triangles, giving 7 polygons with a total of 41 vertices (counting repetitions, as would be necessary in OpenGL polygon calls). Now suppose this cell is subdivided into the 6 tetrahedra shown in figure 4*b.* Four of these tetrahedra require 4 triangles for hardware rendering, and two of them require 3, giving 22 polygons with a total of 66 vertices. Although OpenGL triangle fans can reduce the latter vertex count, the basic polygon and geometry overhead is significantly larger for the subdivided case. The rasterization effort is also higher, since most pixels in the projection of the original hexahedron are covered by at least 3 of these triangles, and those in the projection of quadrilateral *ACGE* are covered by 4.

Therefore, we subdivide only the cells which are intersected by a viewing ray in more than one segment. We assume that the distortion is mild enough so that this can happen only when the viewing ray intersects a single curved face more than once.

For a cell *A* and a specific viewpoint *V,* consider a quadrilateral face *F* divided into two triangles *S* and *T* by a diagonal from the vertex *L,* as shown in figure 3. A viewing ray from *V* can exit *A* through *S* and enter *A* again through *T* if an only if the outward normal to *S* has a positive dot product with the vector *VL,* the outward normal to *T* has a negative dot product with *VL,* and both *V* and the interior of triangle *S* lie on the same side of the plane of triangle *T.* In this case, we call *F* a *problem face,* and *A* a *problem cell*. Our strategy is to think of all quadrilateral faces as in principle already divided into two triangles by the diagonal from their smallest vertex, but to actually project as quadrilaterals the non-problem faces.

The problem cell *A* described above is temporarily subdivided into tetrahedra. However, the cell *B* on the opposite side of *F* need not necessarily be subdivided. Instead, the quadrilateral face *F* of *B* is temporarily replaced by the triangles *S* and *T* for the purpose of sorting and projection from the current viewpoint. If *B* is later determined to also be a problem cell, due to a different problem face, the rest of *B* can be subdivided consistently, as shown in section 4 above.

We sort by the BSP-XMPVO algorithm of [16]. All exterior non-planar quadrilaterals are permanently subdivided into triangles, for use in constructing an accurate BSP tree during preprocessing. For each new viewpoint, all cells are examined to check whether they are problem cells,

and if they are, the temporary changes discussed above are made. Then the visibility sort is run on the modified data structure.

We wanted our algorithm to handle Lagrangean meshes, which can deform (change geometry) during a simulation, and even be remeshed (change topology) when necessary to preserve numerical accuracy. Thus there are several things which can change when interactively viewing a sequence of time steps in a simulation. Listing in order of decreasing impact on the sorting and rendering computation, they are: (*a*) the mesh topology, (*b*) the mesh geometry, (*c*) the viewpoint, (*d*) the scalar data to be visualized, and (*e*) the transfer functions used to assign color and transparency to the scalar data.

Our file format for storing the meshes does not include the topological information necessary to build the adjacency graph, so if the mesh topology changes, we must search for the adjacent faces and build the graph. If the geometry changes, we must recompute the face plane equations, decide which quadrilateral faces on each cell are non-planar and could potentially cause problems, and recompute the BSP tree. If the viewpoint moves, we must redetermine the directions of the edges in the adjacency graph, and determine which cells are problem cells. The problem cells are subdivided into tetrahedra, and those that are no longer problems are restored to their original state. The BSP-XMPVO sort then lists the cells in the correct order for compositing. If only the scalar data or the transfer function changes, the sorted list can be reused in the rendering.

We organized the data structure for the faces so that it can be easily updated if a problem cell is subdivided. Each quadrilateral face has two triangular subfaces, which are determined as soon as the mesh topology is known. Each subface has room for its plane equation, and for two cell pointers, to the two cells that share it (unless it is an exterior face, with only one such cell). If a problem cell is subdivided into tetrahedra, these cell pointers are revised to point to the new tetrahedra. The new tetrahedra have parent pointers to the cell they subdivide, to help in rerouting the cell pointers when a subdivided cell is restored to its original state. We keep the subdivisions for the previous frame, and only add or delete the subdivision of cells whose “problem” state changes.

The data structures for the faces of the original cells are stored in two large linear arrays, one for triangles and one for quadrilaterals. These arrays are accessed in order when the new viewpoint is substituted into the plane equations to determine the graph edge directions, and which cells must be subdivided. This increases cache coherence.

The new tetrahedra for a subdivided cell, and their new separating faces, interior to the original cell, are stored together in a block of memory, to provide memory locality. There are four sizes of blocks, containing 2, 3, 5, or 6 tetrahedra, for the four subdivision cases discussed in section 4. These are allocated in four large arrays, with lists of the used and unused blocks in each array. When a cell no longer needs subdivision, its block is put in the unused list, and can be reused.

In addition to the sorting problem mentioned above, there are other reasons to subdivide a cell into tetrahedra. If volume rendering is combined with surface rendering of contour surfaces, the volume cells must be subdivided by the contour surfaces for correct visibility sorting. As discussed in [7], this is easier to do for tetrahedra, because the all contour surfaces intersecting a tetrahedron do so along parallel planar slices, dividing the volume into parallel slabs. Therefore, we also subdivide into tetrahedra all cells containing contour surfaces. Since a visibility sort of the slabs into which a single tetrahedron is sliced is trivial, only the tetrahedra are placed into the main visibility sort. The semitransparent contour surface polygons are rendered in order between the slabs they separate.

A related situation occurs with non-linear transfer functions, which are necessary to accentuate ranges of the scalar variable of particular interest in the visualization. We use piecewise linear transfer functions for color and opacity, which are linear on the ranges between successive *breakpoint* values of the scalar variable being visualized, but have discontinuities in the value and/or slope at the breakpoints. (See figure 5.) Since the hardware interpolation of color and opacity is only a good approximation when the transfer functions are linear, the cells must be subdivided along the contour surfaces at the breakpoint values. Again, we do this by first dividing the cells containing breakpoint values into tetrahedra, and then into slabs across which the color and opacity varies linearly.

Figure 6, reprinted from [7], shows the coolant velocity magnitude from a 12,936 cell finite element simulation of coolant flow inside a component of the French Super Phoenix nuclear reactor. It was produced for a new viewpoint by the sorting and hardware polyhedron projection methods discussed here in 0.95 seconds on a SGI Power Onyx with one 180 MHZ R5000 CPU, after the adjacency graph and BSP tree were built in 4.92 seconds.

6. Future work

We hope to integrate the cell slicing by contour surfaces or transfer function breakpoints into our current sorting and compositing scheme. This could be done by subdividing the sliced cells into tetrahedra, and revising the directed adjacency graph to account for this additional subdivision, as described above. However, unlike the situation for problem cells, this subdivision is not required for the visibility sort. Thus, if there is no change in topology, geometry, or viewpoint, it should be more efficient to use the main visibility sort, and use a separate small visibility sort for the tetrahedra within such a subdivided cell.

We have optimized the algorithm to subdivide cells only when necessary, in order to simplify the job of the hardware rendering pipeline, but we have introduced more complicated data structures, which put more burden on the CPU. In addition, it takes time to determine the subdivision of the image plane by the projected edges of the zoo elements. We hope to parallelize some of this work on the multiple CPUs of our visualization engine. In particular, when the BSP-XMPVO sort

is executed, it can produce layers of cells, which are all known not to obscure each other. Multiple CPUs could independently divide these cells into polygons and insert the polygons into the rendering pipeline in any order, synchronizing only when each layer is complete.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-ENG-48, and specifically supported by the Accelerated Strategic Computing Initiative. The paper was improved by comments and suggestions of the reviewers.

References

[1] Nelson Max, “Optical Models for Direct Volume Rendering”, IEEE Transactions on Visualization and Computer Graphics”, Vol. 1, No. 2, (June 1995) pp. 99 - 108.

[2] Brian Cabral, Nancy Cam, and Jim Foran, “Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware”, Proceedings, 1994 Symposium on Volume Visualization, ACM Press, pp. 91 - 98.

[3] R. Osborne, H. Pfister, H. Lauer, N. McKenzie, S. Gibson, W. Hiatt, and T. Ohkami, EM-Cube: An Architecture for Low-Cost Real-Time Volume Rendering, Proceedings of the Siggraph/ Eurographics Workshop on Graphics Hardware, Los Angeles, CA, August 1997, pp. 131-138.

[4] Roni Yagel, David Reed, Asish Law, Po-Wen Shih, and Naeem Shareef, “Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing, Proceedings, 1996 Symposium on Volume Visualization, ACM Press, pp. 55 - 62.

[5] Michael Garrity, “Raytracing Irregular Volume Data”, Computer Graphics, Vol. 24, No. 5 (November 1990) pp. 35 - 40.

[6] Peter Williams and Nelson Max, “A Volume Density Optical Model”, Proceedings, 1992 Workshop on Volume Visualization, ACM Press, pp. 61 - 68.

[7] Peter Williams, Nelson Max, and Clifford Stein, “A high accuracy volume renderer for Unstructured Data,” IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 1 (1998) pp. 37 - 54.

[8] Peter Shirley and Alan Tuchman, “A Polyhedral Approximation to Direct Scalar Volume Rendering”, Computer Graphics, Vol. 24, No. 5 (November 1990) pp. 63 - 70.

[9] Jane Wilhelms and Alan Van Gelder, “A Coherent Projection Approach for Direct Volume

Rendering", Computer Graphics, Vol. 25 No. 4 (July 1991) pp. 275 - 284.

[10] Bruce Baumgart, "Winged-edge Polyhedron Representation", Technical Report STAN-CS-320, Computer Science Department, Stanford University, 1972.

[11] Joseph O'Rourke, "Computational Geometry in C", Cambridge University Press, 1995.

[12] Cliff Stein, Barry Becker, and Nelson Max, "Sorting and Hardware Assisted Rendering for Volume Visualization", Proceedings, 1994 Symposium on Volume Visualization, ACM Press, pp. 83 - 90.

[13] Mark Segal and Kurt Akeley, "The OpenGL Graphics System: A Specification", Silicon Graphics, Inc., 1998.

[14] Herbert Edelsbrunner, "An Acyclicity Theorem in Cell Complexes in d Dimensions", Proceedings of the ACM Symposium on Computational Geometry (1989) pp. 145 - 151.

[15] Claudio Silva, Joseph Mitchell, and Peter Williams, "An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes", Proceedings, 1998 Symposium on Volume Visualization, ACM Press, pp. 87 - 94.

[16] Joao Comba, James Klosowski, Nelson Max, Joseph Mitchell, Claudio Silva, and Peter Williams, "Fast Polyhedral Cell Sorting for INteractive Rendering of Unstructured Grids", Proceedings of Eurographics '99, Computer Graphics Forum, Vol. 18, No. 3 (September 1999).

[17] Peter Williams, "Visibility Ordered Meshed Polyhedra", ACM Transactions on Graphics, Vol. 11, No. 2, (April 1992) pp. 103 - 126.

[18] Martin Newell, R. Newell, and T. Sancha, "Solution to the Hidden Surface Problem", Proceedings of the ACM National Conference (1972) pp. 443 - 450.

[19] Martin Newell, "The Utilization of Procedure Models in Digital Image Synthesis", Ph. D, Thesis, University of Utah (1992), report UTEC-CSc-76-218.

[20] Gregory Nielson and Junwon Sung, "Interval Volume Tetrahedralization", Proceedings of IEEE Visualization '97, pp. 221 - 228.

[21] Albertelli, Guy, and Roger Crawfis, "Efficient Subdivision of Finite-Element Datasets into Consistent Tetrahedra," Proceedings of IEEE Visualization '97, IEEE Computer Society Press (1997) pp. 213 - 219.

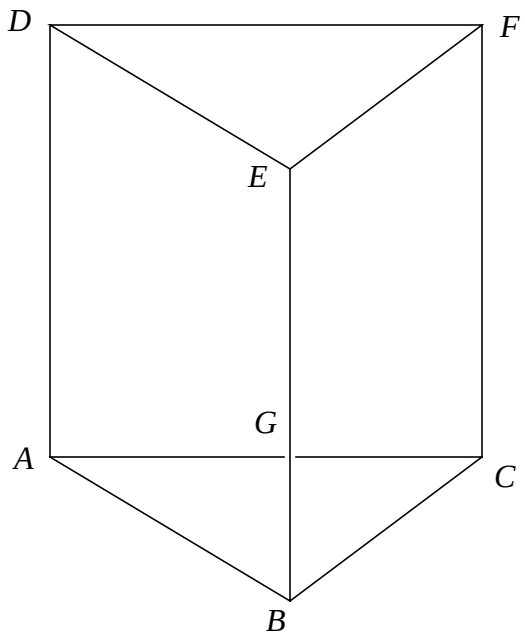

Figure 1. Image plane projection of a prism.

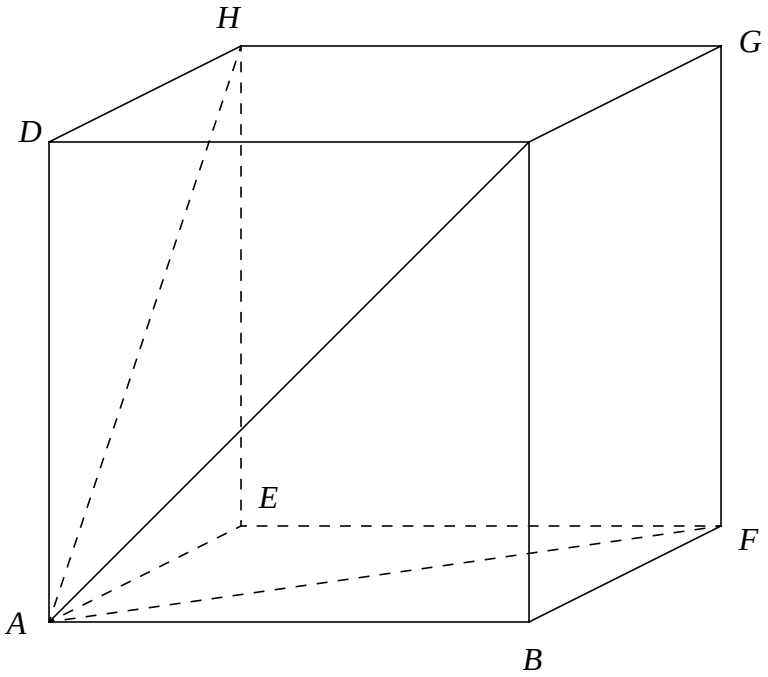

Figure 3. A cube, with three face diagonals from vertex A.

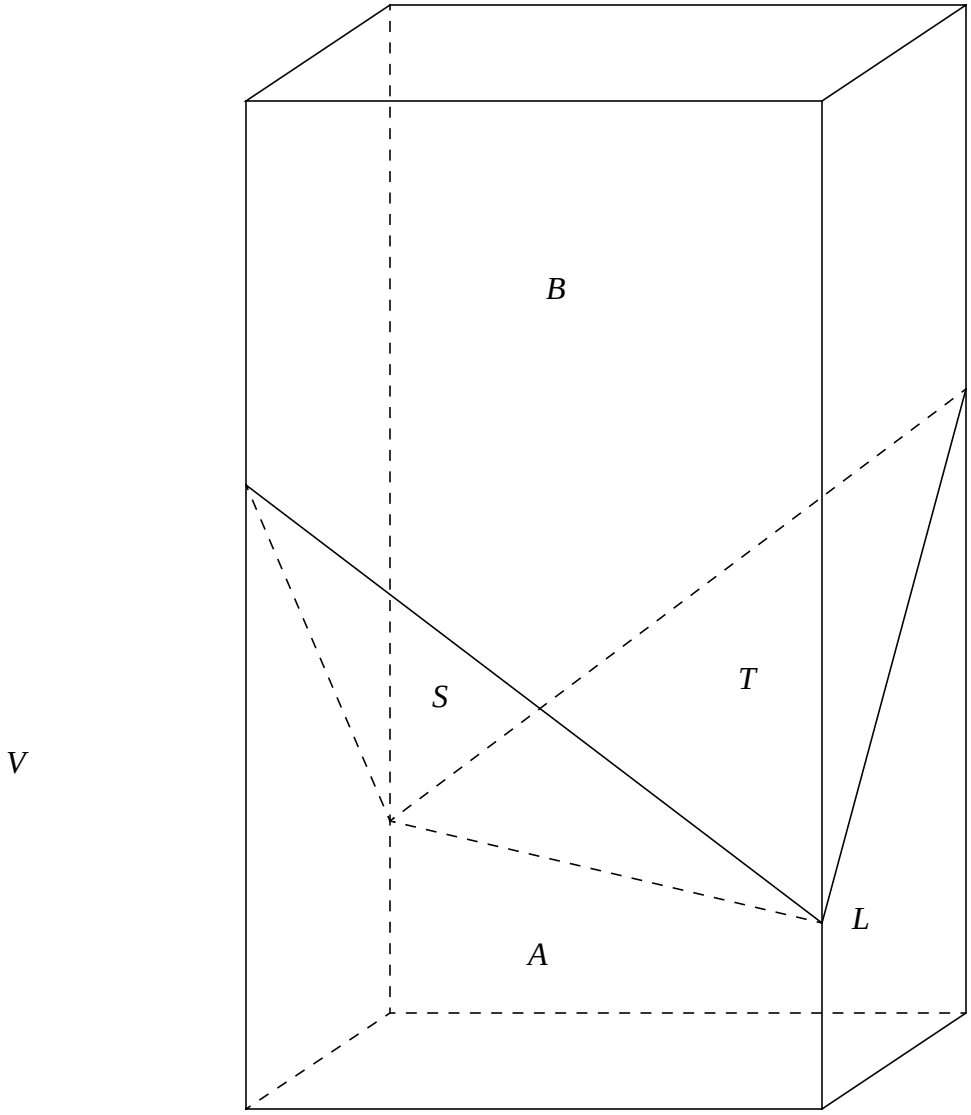

Figure 2. A non-planar face between cells *A* and *B*, approximated by triangles *S* and *T*.

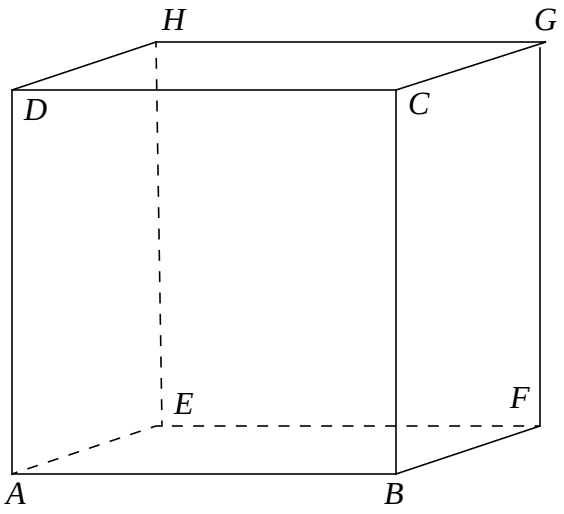

Figure 4*a*. Projection of a cube.

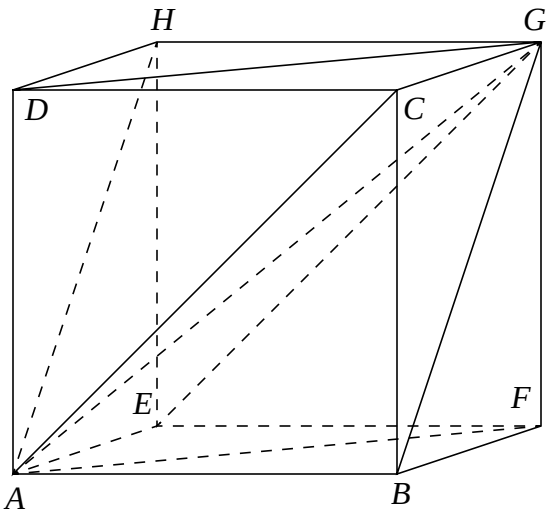

Figure 4*b*. Cube divided into 6 tetrahedra.

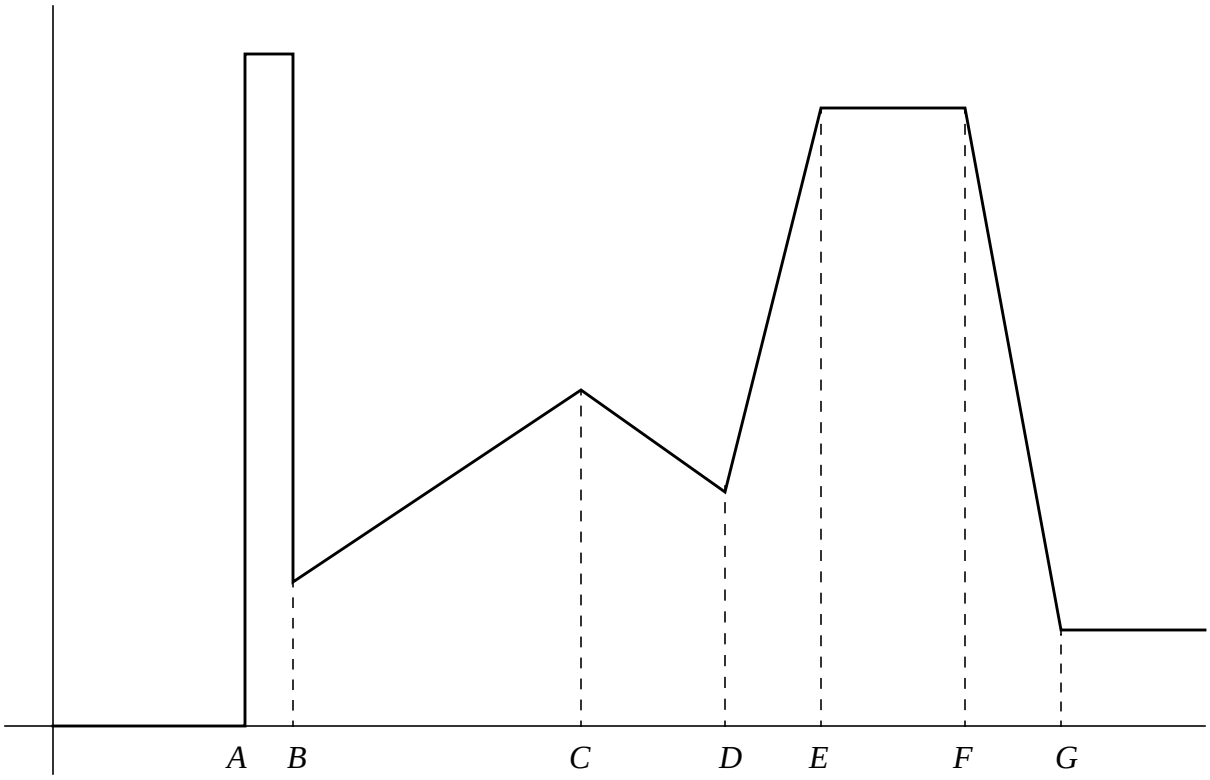

Figure 5. Breakpoints *A*, *B*, *C*, *D*, *E*, *F*, and *G* in a piecewise linear transfer function. Note the discontinuities at breakpoints *A* and *B*, placed to emphasize a particular range of values.